UNITÉ DE RECHERCHE
NRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N°785

UN CONFERENCIER REPARTI SOUS SOS ET PORTAGE DE L'EDITEUR DE LIENS DYNAMIQUE

Yvon GOURHANT

JANVIER 1988

Un Conférencier réparti sous SOS et portage de l'éditeur de liens dynamique
A distributed electronic conference and port of a dynamic link editor under SOS

Yvon Gourhant
Institut National de Recherche en Informatique et en Automatique
Domaine de Voluceau
Rocquencourt
B.P 105, 78153 Le Chesnay Cedex, France
uucp: gourhant@corto.inria.fr

4 janvier 1988

Résumé

Ce rapport développera deux études indépendantes, reposant sur le système à objets réparti SOS.

Tout d'abord, une application au-dessus de SOS, un "conférencier réparti", exposera les problèmes liés aux situations multi-utilisateurs. L'objectif de ce "conférencier réparti" consiste à expérimenter la première version de SOS, à montrer ses aspects objet et distribué, ainsi que la mise en œuvre des mécanismes de mandataire.

La réalisation de ce conférencier permettra de démontrer certaines insuffisances de SOS version 1, qui seront corrigés dans la version 2.

Ensuite, l'étude approfondie de l'"éditeur de liens dynamique" et de son portage sous UNIX [1] Système V au format COFF présentera une partie de SOS et introduira les problèmes de liaisons dynamiques dans Unix, selon les versions. Ce portage est la première étape du portage de SOS sur la machine cible. Ce travail a permis d'entrevoir des solutions aux futures tâches à réaliser, que nous détaillerons.

[1] Unix est une marque déposée des laboratoires Bell

Abstract

The first part of this report discusses the realization of a distributed electronic conference, on top of SOS. We will expose problems related to multi-user situations. The goal of this application was to test the distribution support mechanisms, provided by the first version of SOS. It was also an example of an application, structured using the Proxy Principle. This realization established the lack of certain functionalities, fixed and provided by the second version of SOS.

The second part concerns the dynamic link editor and its adaptation to the Unix System V COFF format. This work is the first step to the port of SOS to the target machine.

Remerciements

Je tiens à remercier toute l'équipe du projet SOR à l'INRIA, pour m'avoir acceuilli chaleureusement.

Je remercie M. Shapiro, qui m'a initié aux Systèmes à Objets Répartis, et pour ses précieux conseils dans l'élaboration de ce rapport.

Je remercie également P. Gautron, l'auteur de l'éditeur de liens dynamique sous SOS, pour son encadrement et ses judicieux propos sur les liaisons dynamiques.

Remarque

La syntaxe et les fonctionnalités utilisées dans ce document correspondent à SOS Prototype Version 1 et sont succeptibles de modification dans les versions ultérieures.

Chapitre 1

Introduction

Le support des deux études présentées ici est le système à objets réparti SOS ("SOMIW Operating System") [Sha86b], réalisé par le projet SOR (Systèmes à Objets Répartis) situé à l'INRIA-Rocquencourt. SOS est la composante système d'exploitation du projet européen Esprit 367 SOMIW ("Secure Open Multimedia Integrated Workstation"). Les objectifs de SOMIW sont la conception et la réalisation d'un environnement bureautique intégré pour le traitement de documents multi-média. Le rôle de SOS est de fournir une interface structurée permettant de traiter des documents contenant indifféremment des données textuelles, graphiques, vocales ou images animées.

Les deux études présentées ici sont :

Un "conférencier réparti", concu afin d'exercer les mécanismes de SOS.

Le portage de "l'éditeur de liens dynamique" au format COFF, dans le cadre du portage de SOS sur le Métaviseur, machine cible du projet.

1.1 Présentation de SOS

SOS [Sha87a] est un système à objets réparti reposant sur la notion de "mandataire" [Sha86a]. La version de SOS étudiée est la version 1 [Sha87b] livrée en février 87 aux partenaires du projet SOMIW. Ce prototype tourne au-dessus d'UNIX 4.2BSD [Ker84] sur stations Sun. Cette maquette est écrite en C++ [Str85], permettant d'allier les avantages des langages à objets à l'efficacité d'un langage compilé.

L'objectif de SOS n'est pas de cacher la répartition, mais d'offrir des interfaces structurées, faisant abstraction du type de données traitées. Cette structuration se base sur une organisation de l'espace d'adressage sous forme de "domaines". La communication à l'intérieur d'un domaine est libre. Les communications entre domaines s'effectuent selon le principe des *mandataires* (cf. 2.4). Un domaine élémentaire est soit un espace de mémoire virtuel, soit un en-

semble d'objets coopérants, dénommés "groupe d'objets". Les relations entre domaines sont du type Client/Ressource. Les ressources sont implémentées par des *services*, dont le rôle est essentiellement d'exporter un représentant de cette ressource (`mandataire`). SOS sera expliqué plus en détail au chapitre 2.

1.2 Speak : Un conférencier réparti

L'application `speak` a pour but, de démontrer la possibilité d'utilisation du principe de mandataire et des fonctionnalités de SOS. Le sujet de cette application, un conférencier réparti, a été choisi parce qu'il allie une fonctionnalité très simple à une structure de communication représentative des applications multi-utilisateurs en général. C'est pourquoi les fonctionnalités graphiques et l'interface utilisateur ont été délibérément négligées.

1.2.1 Utilisation

L'annexe A représente quelques étapes d'un petit exemple avec trois utilisateurs. Tout d'abord, nous descrivons l'établissement de l'environnement de l'application `speak`. Cet environnement consiste en deux parties, le noyau SOS et le serveur `speakServer`. Puis, l'utilisateur Gourhant lance l'application `speak` et attend la venue d'un nouvel arrivant. L'utilisateur Shapiro se joint à l'application, suivi par Gourhant, à nouveau mais sur un autre terminal. Chacun peut alors communiquer avec n'importe quel autre utilisateur. Nous détaillons ces étapes ci-dessous.

1.2.2 Création de l'environnement

```
% sos &
```

Cette commande se charge de lancer un processus Unix qui simule le noyau SOS.

```
% speakServer &
```

Cette commande permet de lancer explicitement le *serveur* de l'application : `speakServer`. (Ceci pourrait aussi être fait de manière automatique lors du lancement du noyau). Dans SOS Version 1, chaque *serveur* doit s'enregistrer dans le *service de Nommage*, et doit être actif lors d'une demande d'un *client* (pas de *service de Stockage*). Le rôle de ce *serveur* est essentiellement d'exporter des *mandataires* vers les *clients* (cf. section 3.4). La gestion d'une copie de la table des utilisateurs chez le *serveur* `speakServer` permet de montrer que le *serveur* peut contenir ses propres données, (en particulier, les informations privées).

1.2.3 Lancement de l'application speak

Le système et son environnement lancés, chaque utilisateur peut lancer l'application speak indépendamment des autres utilisateurs.

```
% speak
```

Cette commande permet de lancer le programme ''speak'' afin d'entamer une conférence. Le programme se charge pour lui, de demander un *mandataire* au *serveur* speakServer et d'appeler les méthodes associées. Lorsqu'un second utilisateur effectue la même démarche, le *serveur* speakServer lui délivre un autre mandataire, et cet utilisateur peut alors se joindre à la conférence. La situation est alors entièrement symétrique. Un nombre quelconque d'utilisateurs peut se joindre à cette même conférence. Chaque utilisateur peut alors envoyer et recevoir des messages aux autres membres présents. L'envoi de messages peut aussi être diffusé à tous les utilisateurs. Les dernières lignes de l'exemple d'utilisation, présenté en annexe A, illustrent l'utilisation du mécanisme d'exception implémenté dans SOS, permettant de différencier le lieu de détection d'un événement et le lieu de sa signalisation.

Dans le chapitre 3, consacré à cette application, nous présenterons les structures externes et internes de speak, ses limitations avec leur justification. Enfin, nous présenterons brièvement la structure finale proposée pour SOS version 2.

1.3 Portage d'un éditeur de liens dynamique au format COFF

L'édition de liens dynamique dans un environnement objet compilé paraît un atout important. Les avantages obtenus sont proches de ceux liés aux langages interprétés (Smalltalk, Lisp), en conservant l'efficacité des langages compilés. Par ailleurs, SOS effectue la migration d'objets s'accompagnant de la migration de leur code, ainsi que la liaison dynamique dans le contexte importateur.

1.3.1 En quoi consiste l'édition de liens dynamique

L'édition de liens dynamique consiste à charger dynamiquement du code et des données externes pendant l'exécution d'un programme. Une première phase statique est nécessaire, afin de préparer l'exécutable à reconnaître le moment d'importer dynamiquement le code absent. La seconde phase, lors de l'exécution, consiste à charger le code absent et d'effectuer la relocalisation d'informations par rapport à son adresse de chargement effective.

Dans la version réalisée pour SOS, la première phase se situe à la compilation et à l'édition statique, et la seconde est réalisée par une bibliothèque liée statiquement avec l'exécutable.

1.3.2 Réalisation

Le chapitre 4 présentera deux exemples de liaisons dynamiques. Le premier est inspiré du code de Franz Lisp de B. Joy (Université de Berkeley), pour exécuter des modules compilés externes à l'environnement de l'interpréteur. Le second, Camphor, sera décrit plus en détail, car la version pour SOS s'en inspire. Nous décrirons donc ses insuffisances et les modifications apportées dans SOS.

La seconde partie du chapitre 4 sera consacrée aux modifications apportées lors du portage, des problèmes rencontrés et de leurs résolutions justifiées.

Chapitre 2

Présentation de SOS

2.1 Comparaison avec d'autres projets de recherche dans ce domaine

SOS se distingue des autres systèmes répartis en situant ses objectifs entre les systèmes basés sur la communication par messages et les systèmes à transactions. Les systèmes à messages tels que Chorus [Zim84], Accent [Ras81] ou le V-System [Che84] reposent sur la transparence du réseau. Mais leur simplicité est contrebalancée par l'absence de structuration. En effet, ces systèmes disposent d'un adressage plat, et ne propose aucun typage. Les systèmes tels que Argus [Lis83], Isis [Bir85], ou Clouds [McK85] ont, contrairement aux précédents, l'avantage d'être structurés mais perdent en flexibilité et en efficacité.

L'approche SOS est une approche intermédiaire :

- domaines structurés et contrôles inter-domaines
- efficacité (effets de localité) et encapsulation dans la notion de mandataire
- transparence implémentée dans les mandataires
- cohérence et dépendances entre objets : groupes

Les objectifs de SOS sont proches de ceux d'Emerald [Bla86] : offrir des interfaces de hauts niveaux bien structurées. Mais tandis que Emerald encapsule ses interfaces dans le langage, SOS fournit des outils systèmes, donc moins transparents.

La transparence dans le système Apollo/Domain [Lea83] est fournie par les services de haut niveau, implémentés au-dessus d'un adressage unique à tout le réseau. Domain est un système intégré :

- Chaque service est indépendant de sa localisation

- Chaque station est autonome
- Tous les utilisateurs possèdent une vue uniforme du système [Alm85] [Wal83]

Le système Domain consiste en un ensemble de nœuds interconnectés par un réseau local. Ces nœuds peuvent être soit du type client, soit du type serveur. Chaque nœud possède un translateur dynamique d'adresses permettant d'obtenir un espace d'adressage unique à tout le réseau. Des services viennent fournir des interfaces de plus haut niveau. Ainsi, le plus bas niveau d'interface entre les applications et les entrées/sorties, assuré par le service de Stockage, est du type "stockage à un seul niveau" (single-level store). Le niveau supérieur présente quand à lui un ensemble de procédures appelées "standard streams". Le service de Nommage et le service de Verrouillage fournissent des outils d'abstraction à partir de l'identification unique de ces objets (UID).

Dans Clam [Cal87] comme dans SOS, le système ne cherche pas à cacher la répartition, mais à offrir des outils pour structurer les applications. Une notion commune à ces deux systèmes est la notion d'encapsulation d'interface. En effet, la liaison et le chargement de modules externes est dynamique. L'objet appelant ne connaît de la ressource que l'objet importé, intégrant les notions de protection, de numéro de version, et de révision. Ce modèle, différent de l'approche traditionnelle Client/Serveur, permet d'interfacer les programmes clients et les ressources de facon symétrique.

Dans Clam, la structuration en couches s'applique aux interfaces graphiques, non seulement à celles de type "sortie" (support de plusieurs types de gestionnaires d'affichage), mais aussi aux interfaces de type "entrée" (clavier/souris). Ces dernières soulèvent des problèmes plus difficiles, tels que le traitement d'événements asynchrones et de propagation des signaux aux différents modules ou couches. Cette structuration en couches facilite le parallélisme entre applications et à l'intérieur des applications elles-mêmes sous forme de "mini-processus".

2.2 La notion d'objet

Un objet [Jon79] est une entité doublement structurée : la représentation interne contient les données de l'objet et son implémentation ; l'interface externe présente les méthodes d'accès manipulant ces données.

Un objet SOS est défini par une classe C++. Chaque classe contient une liste de *méthodes* qui permettent à ses instances de répondre aux invocations qui leur sont effectuées. L'objet commence son existence à l'instanciation de cette classe, lors de l'appel de la méthode appelée "constructeur", définie implicitement (allocation mais pas d'initialisation), ou explicitement (allocation et initialisation par l'utilisateur).

2.3 L'approche objet

Les caractéristiques de l'approche objet sont :

- La modularité

  L'interface externe d'un objet est bien définie et localisée dans la définition de la classe de l'objet, et son implémentation est cachée au client.

- Les types de données abstraits [Lis82]

  L'utilisation par d'autres modules (encapsulation d'interface)

- La généricité

  La généricité permet de définir un objet sans définir complètement son type, et de ne préciser ce type qu'au moment de l'instanciation. (La généricité n'existe pas explicitement dans C++, mais est possible par macro définition).

- L'extensibilité

  Un programmeur d'application peut créer ses propres classes qui ont un statut égal aux classes prédéfinies.

- La hiérarchie des types.

  Dans C++, plusieurs classes *dérivées* définissent une hiérarchie. Une classe *dérivée* hérite des champs et méthodes de sa classe mère. Ainsi dans SOS, un objet sera reconnu du système, si la classe qui le définit, dérive de la classe mère `sosObject`, définie par le système. Lors de son instanciation, les constructeurs de chacune des deux classes sont appelés.

  Les méthodes *virtuelles* de la classe mère sont redéfinissables dans la classe dérivée. Par exemple, dans un contexte musical, la classe "blanche" et la classe "croche" dérivent toutes deux de la classe "note" mais n'implémentent pas de la même facon la méthode "durée".

- évolution et réutilisabilité [Mey87]

- indépendance entre classes (développements séparés)

- différence entre spécification et conception : deux niveaux d'abstraction complémentaires.

2.4 La notion de mandataire

(Voir figure 2.1)

2.4.1 Notion de contexte, d'accointance et d'OID

Les objets de SOS sont "mappés" dans des espaces de mémoire virtuels, dénommés *contextes*. Ces objets sont appelés *"accointances"* de ce contexte. Chaque accointance possède un identifieur unique à tout l'univers SOS. Cet identifieur se dénomme "OID concret".

La connaissance d'un OID ne donne aucun droit sur l'objet identifié. L'obtention de cet OID ne fait que donner la possibilité d'appeler la méthode d'exportation de mandataire `givePxoy` de l'objet identifié. Cette méthode peut décider d'exporter ou de ne pas exporter un mandataire à un client donné. Cette méthode se termine par défaut sur une exception (`''refused''`). Une classe désirant exporter des mandataires doit la redéfinir.

2.4.2 Notion de mandataire et de mandant

La communication entre les objets d'un même contexte s'effectue par de simples appels procéduraux.

Pour accéder à une ressource située hors de son contexte, un client doit d'abord acquérir (importer) un objet *mandataire*, qui représente la ressource auprès du client. Celui-ci ne peut accéder à la ressource que par invocation (locale) du mandataire. Ce mandataire peut soit répondre localement à cette invocation, soit la transmettre à un objet réalisant la ressource dans le contexte du serveur, appelé *mandant*. Le canal de communication entre un mandataire et un *mandant* est appelé *trapReference*. La communication entre un mandataire et un mandant se fait par la primitive d'invocation inter-contexte `crossInvoke`. La sémantique de cette primitive est identique celle de l'appel de procédures distantes. Cette primitive invoque la procédure `stub` [1] de l'objet désigné, en lui passant une copie des paramètres. La procédure `stub` dispatche les appels vers les méthodes qui implémentent la ressource.

2.4.3 La notion de groupe

Chaque accointance possède, outre son OID concret (unique), une liste d'OIDs (non uniques) hérités de la ressource représentée, appelés "OIDs de groupe".

Un groupe est simplement l'ensemble des objets ayant un OID de groupe en commun.

Un objet SOS peut appartenir à plusieurs groupes.

[1] Dans l'état actuel de notre environnement de programmation, le code du mandataire et de la procédure `stub` doit être écrit à la main.

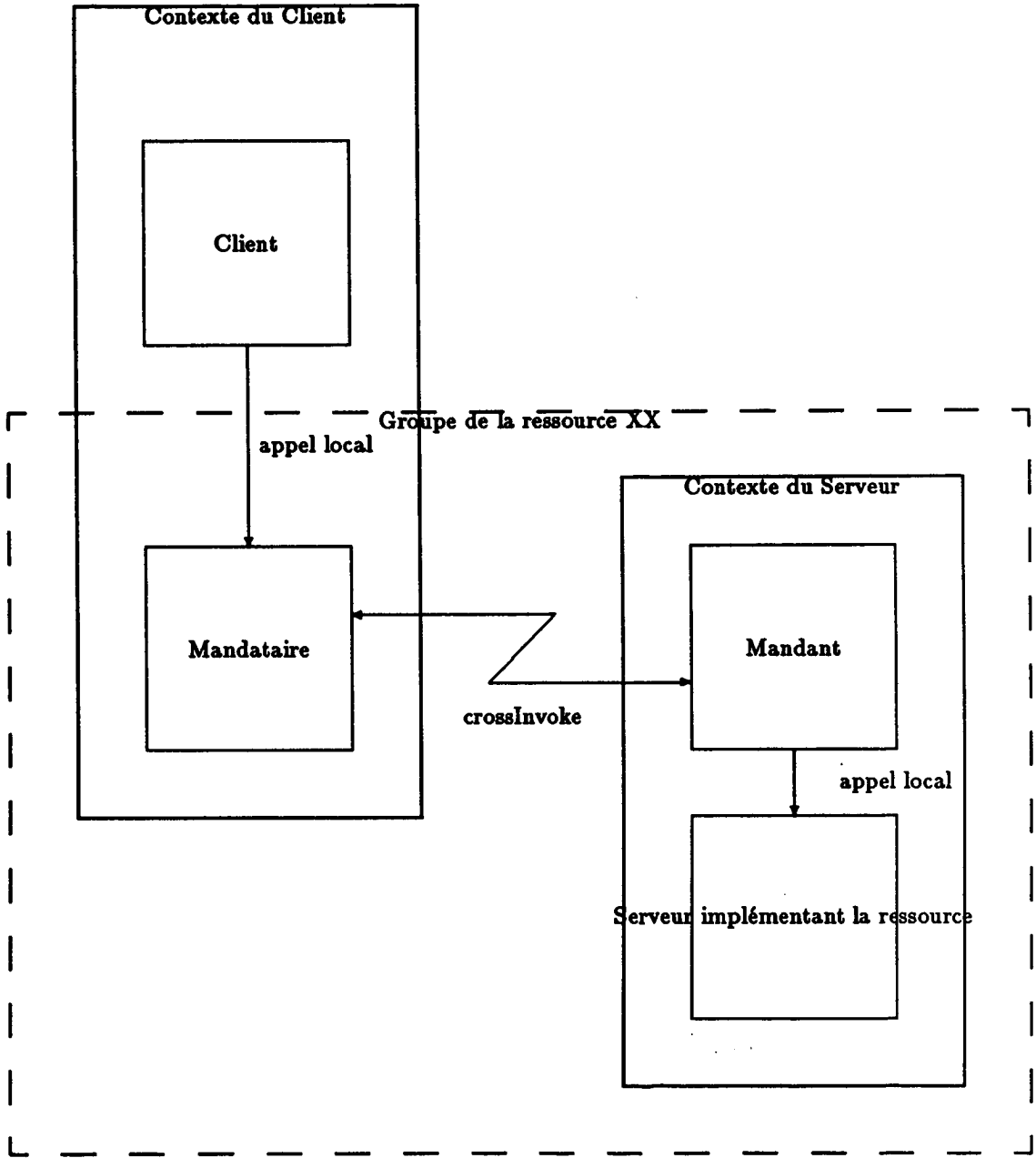


Figure 2.1 : principe du mandataire

2.5 Le noyau et les services système

SOS est un sytème "ouvert" [Lam79], basé sur un petit noyau et des services externes exécutant des fonctions systèmes.

2.5.1 Le noyau

Dans la version actuelle, le noyau est une couche au-dessus d'Unix. Il réalise: la création et la destruction des contextes, la gestion des tâches parallèles dans un contexte, et les communications inter-contexte. Les tâches [Str82], définies sous forme de classes dans la bibliothèque standard C++, peuvent s'éxécuter de manière quasi-parallèle au sein d'un même espace d'adressage. Les objets tâches peuvent se synchroniser à l'intérieur d'un même contexte et communiquent selon le modèle producteur-consommateur. Il existe un séquenceur de tâches dans chaque contexte.

2.5.2 Les services systèmes

Le service d'accointances

Le service d'accointances est le gestionnaire réparti d'objets. Ce service s'occupe de la localisation et de la migration des objets, en collaboration avec les services de stockage et de communication, et gère les `trapReferences`. Chaque contexte possède un mandataire du service d'accointances.

Le service de Nommage

Le service de Nommage établit la correspondance entre un nom symbolique et l'objet qu'il désigne. Chaque contexte possède un mandataire du service de Nommage.

Le service de Stockage

Le service de Stockage permet de stocker et d'extraire des segments sur le disque. Ces segments représentent la forme passive des objets SOS. Ce service est en cours de réalisation.

Le service de Communication

Le service de communications distantes fournit une bibliothèque de protocoles. La sémantique des appels de procédure distante est identique aux communications inter-contextes. Ce service délivre des mandataires selon le concept de "famille", qui élargit la notion de groupes à des objets de sites différents. Le choix du protocole de communication est fait lors de la création de la famille. Ce service n'existe pas dans SOS Version 1.

Chapitre 3

Un conférencier réparti : speak

Après avoir montré l'utilisation de **speak** dans l'introduction, et les mécanismes de base de SOS dans le chapitre précédent, ce chapitre va décrire les modules de **speak**. Cet exemple nous permettra d'expliquer les causes des insuffisances actuelles, et leur résolution sous SOS version 2.

3.1 Structure de l'application

La structure de ce conférencier diffère de celle du programme similaire **talk** d'Unix 4.2BSD, tout du moins dans la conception.

3.1.1 "Talk", un exemple à la Unix

Talk est un programme concu de manière centralisée, où les processus communiquent de manière directe, à l'aide de connecteurs ("sockets"), qui permettent la communication entre deux processus qui n'ont pas de relation parentale (au sens Unix). La version multi-machines introduit, entre ces deux processus, un "démon" intermédiaire : **talkd**, lancé sur chaque station, et qui est à l'affût de toutes les demandes adressées à un utilisateur de sa propre station.

Ainsi, lorsque l'utilisateur X sur la station A veut discuter avec l'utilisateur Y sur la station B, la connexion entre les deux utilisateurs s'établit ainsi :

Le programme **talk** sur la station A, envoie une demande au serveur **talkd** sur sa propre station A.

Le serveur **talkd** de la station A se connecte avec **talkd** sur la station B en lui donnant en paramètre l'adresse de sa propre "socket" (utilisée pour établir cette connexion).

Le programme **talkd** sur la station B, annonce à l'utilisateur Y que X veut communiquer et agit alors comme un point de rendez-vous entre les deux utilisateurs.

Lorsque Y, averti par ce message sur son terminal, appelle le programme **talk** pour discuter avec X, c'est la même "socket" qui est alors utilisée et les deux partenaires peuvent communiquer de manière symétrique.

La démarche suivie pour réaliser **speak** est inverse. La répartition est prévue dès la conception. Mais paradoxalement, la version réalisée se rapproche fortement de **talk**, à cause des limitations de la version 1 de SOS. En effet, l'absence de connexion multipoints ne permet pas la réalisation du modèle réparti initialement conçu. Finalement, ce modèle ne sera réalisé que sous SOS version 2.

3.2 Les contraintes du modèle "multi-utilisateurs"

Les logiciels répartis multi-utilisateurs servent généralement à mettre en évidence la transparence au réseau ("network tranparency"). Ainsi, le jeu Amaze [Ber84] pour le V-Kernel.

Comme déja noté, l'objectif de SOS n'est pas de cacher la répartition, mais de fournir des outils pour structurer des applications réparties. Dans **speak**, si le type de données traitées n'est plus du texte (interface clavier), mais de la voix ou des images, c'est uniquement l'interface avec l'utilisateur qui subira des modifications.

La démarche structurée de l'application Edmas [Alm84] est similaire à celle suivie pour **speak**. Cette application de courrier réparti sur un réseau local permet de mettre en valeur les avantages du système Eden [Bla85]. Une boîte aux lettres n'est plus un fichier (entité passive), mais un objet réagissant aux messages envoyés par les utilisateurs. La protection est encapsulée dans la transmission de la capacité de l'objet "mailbox", et l'atomicité des opérations est assurée par cet objet lui-même.

L'application **speak** est aussi divisée en plusieurs objets installés dans des contextes. Les relations entre ces objets mettent en œuvre les notions de groupe d'objets et de communications dans un contexte et inter-contextes. La structure réalisée dans SOS version 1 est représentée sur la figure 3.1.

Par exemple, le gestionnaire de terminal (**mytty** sur la figure 3.2) est un objet qui communique par envoi ou réception de messages. La communication entre contextes est encapsulée par le mandataire délivré par le service **speakServer**. Ce mandataire peut, soit traiter localement la demande, soit la transmettre au serveur par le biais de la primitive **crossInvoke()**. Dans SOS version 1, un mandataire ne peut avoir qu'un seul mandant.

Les messages tapés par les utilisateurs sont retirés par le gestionnaire de terminal, qui les transmet au mandataire. Ce dernier analyse le contenu et, si

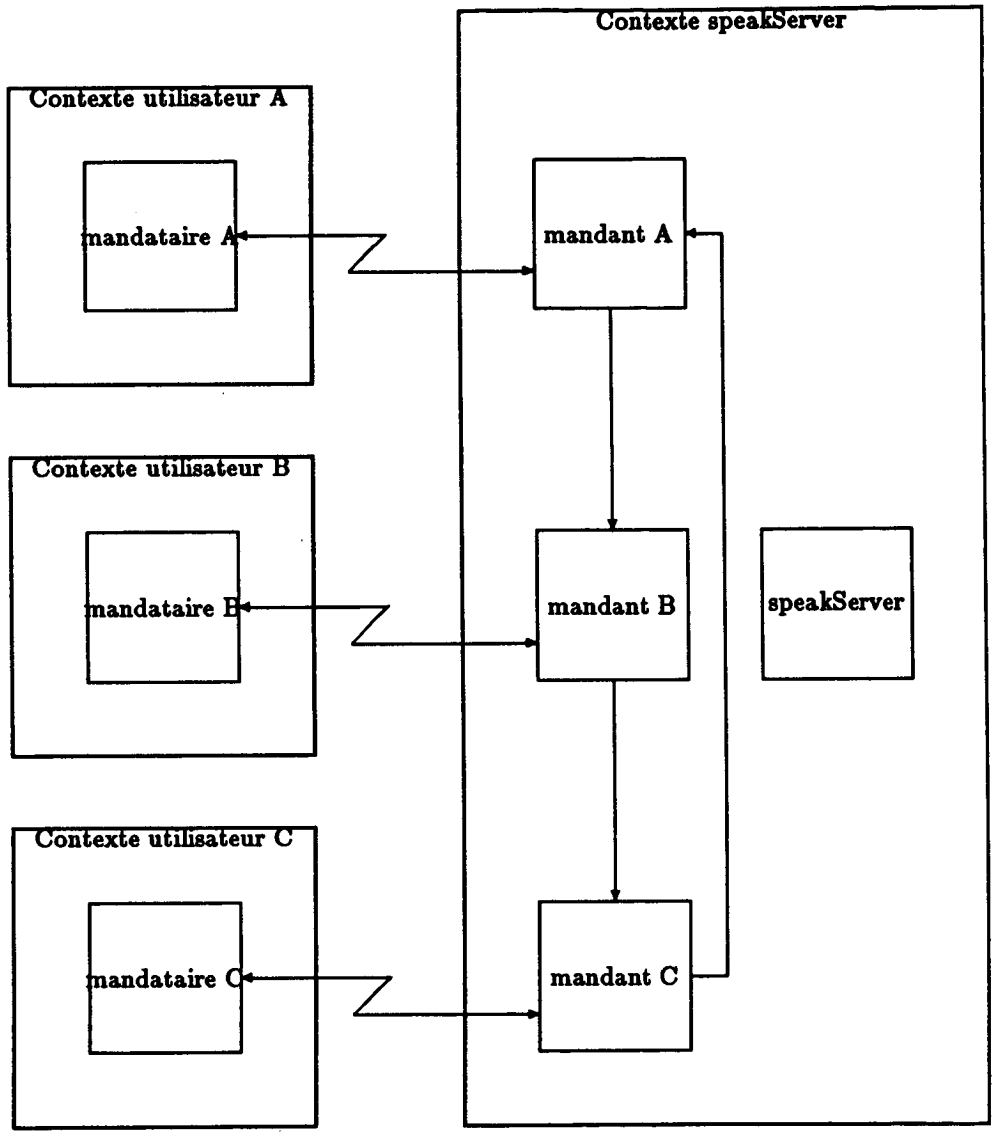


Figure 3.1 : modèle réalisé pour SOS-V1 avec 3 utilisateurs

nécessaire, transmet à son mandant. La communication en anneau est réalisée par les mandants, à l'intérieur du contexte du serveur **speakServer**. Le mandant désigné (par le champs destination du message) effectue une cross-invocation vers son mandataire. Ce mandataire s'occupe alors de placer le message dans la queue de sortie du gestionnaire de terminal. Finalement, le message apparaît sur l'écran de l'utilisateur voulu.

La suite de ce chapitre va détaillller la structure de l'application.

3.3 Le contexte client

Le contexte d'un client se divise en trois parties (figure 3.2) :

- la commande **speak**
- le gestionnaire de terminal **mytty**
- la tâche **tspeak**, qui importe le mandataire du serveur **speakServer** pour relayer les messages.

3.3.1 La commande speak

La routine principale de la commande **speak** effectue les initialisations nécessaires et la division en modules (vérifier que le serveur **speakServer** existe bien dans l'environnement, tester l'ouverture du terminal, lancer les tâches de lecture et d'écriture sur le terminal, lancer une tâche intermédiaire **tspeak** qui importe le mandataire, et enfin synchroniser les différentes tâches).

3.3.2 Le gestionnaire de terminal

Le gestionnaire de terminal est encapsulé dans la classe **mytty**, qui exporte deux queues (flots d'entrées/sorties) et deux méthodes (*open* et *ttyname*, qui retournent respectivement le compte-rendu de l'ouverture du terminal, et le nom symbolique du terminal). L'implémentation de la classe **mytty** associe à chaque queue, une tâche et un **UnixChannel**, afin de ne pas perturber les autres tâches du même contexte[1]. Les **UnixChannel** implémentés dans le noyau SOS, permettent d'effectuer des entrées/sorties non bloquantes et étendent le concept du RPC au "Remote Pipe and Procedure Call" (RPPC).

3.3.3 La tache intermédiaire : tspeak

La tâche **tspeak** importe un mandataire du serveur **speakServer**, et suit le protocole pour se joindre à l'application (cf. 3.4.3), en appelant les méthodes appropriées du mandataire. Une fois la connexion établie, cette tâche intermédiaire

[1] L'utilisation de l'appel système "read" désactiverait toutes les tâches attachées au processus Unix qui exécute ce contexte

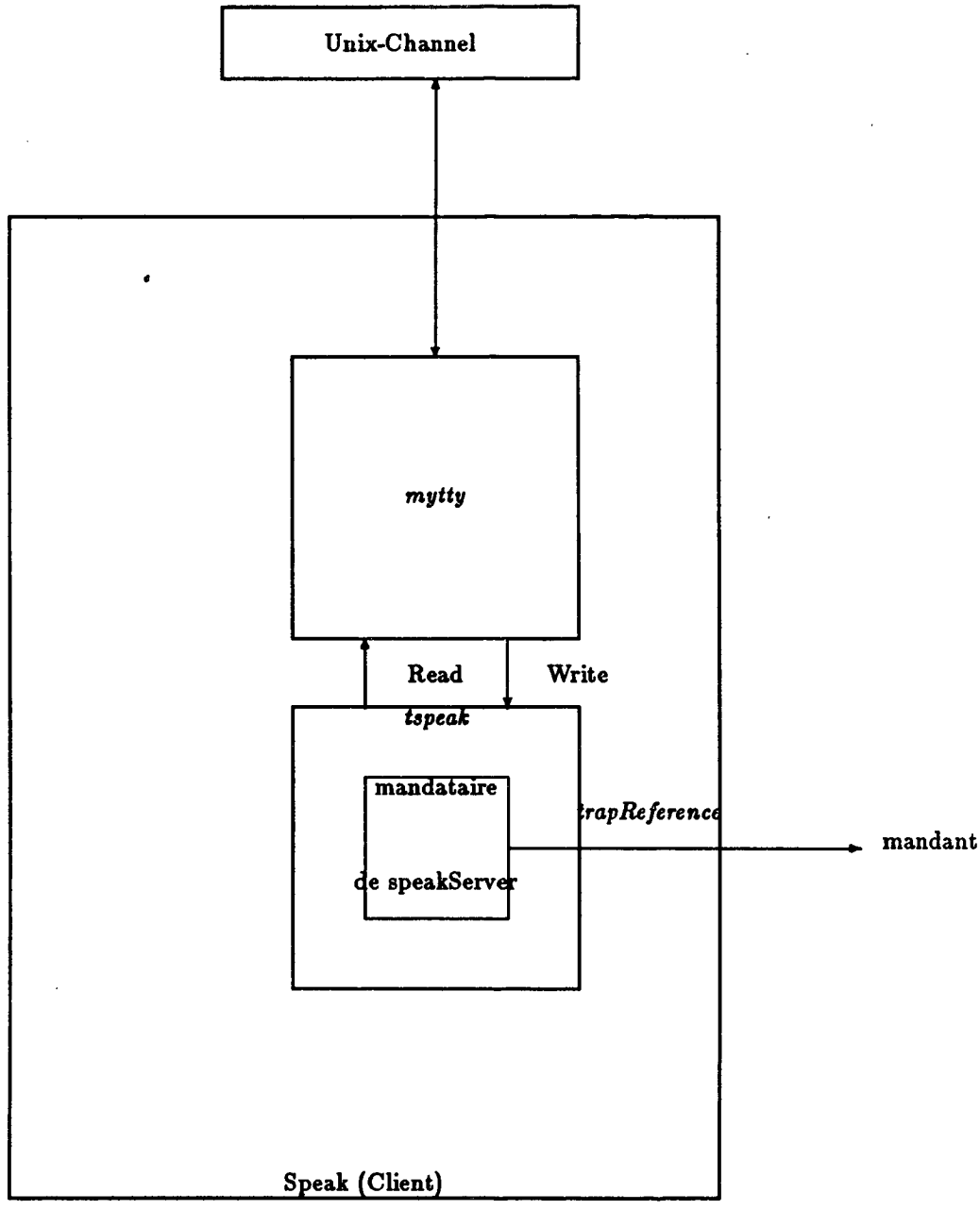


Figure 3.2 : Struture d'un Contexte Speak (Client)

boucle sur le retrait des objets de la queue de lecture du gestionnaire de terminal et leur transmission au mandataire.

La version actuelle de C++ ne permettant pas l'héritage multiple, nous devons créer cette tâche intermédiaire car le mandataire hérite déjà de la classe "sosObject".

3.3.4 Le mandataire speak

Le mandataire offre une interface simple : un protocole d'ouverture, et une méthode d'envoi de messages.

Dans la version 1, les mandataires **speak** ne peuvent communiquer de manière directe. Leur rôle consiste uniquement à vérifier la justesse des messages tapés par les utilisateurs et de les transmettre à leur mandant associé. Pour cela, chaque mandataire tient à jour une copie locale de la table des utilisateurs connectés. Seul un message destiné à l'un de ceux-ci, ou en diffusion, sera transmis au mandant. Cette copie est mise à jour lors de la réception des messages. Lorsqu'un nouvel utilisateur se joint à l'application, le protocole d'ouverture envoie un message à tous les utilisateurs. Cet utilisateur est alors enregistré dans chaque copie locale. La méthode pour quitter l'application est identique, le message étant typé différemment.

Les relations entre la tâche **tspeak** et le mandataire illustrent le mécanisme d'exceptions implémenté au-dessus de C++ dans SOS. En effet, lorsque le format tapé par l'utilisateur est erroné ou que le destinataire n'est pas dans la table des utilisateurs, ou que c'est le caractère de fin, le mandataire génère une exception. Celle-ci est prise en compte par la tâche **tspeak**, qui selon le cas, émettra un message d'erreur, ou terminera proprement l'application. [2]

3.3.5 Déroulement de l'importation d'un mandataire

L'importation d'un mandataire s'écrit de la manière suivante :

```
dynamic ("speakServer") class proxy * sp;
sp = new proxy;
```

Le mot-clé **dynamic** indique que cet objet sera importé dynamiquement. **speakServer** est le nom symbolique du serveur. Enfin "sp" est une instance de la class **proxy**, exportée dans le contexte client par le serveur. C'est seulement lors de l'instanciation (**new proxy**) que cet objet va être importé. Cette demande est relayée au serveur **speakServer** par l'intermédiaire du service d'accointances.

[2] La version 1 de SOS ne gérant pas les dépendances entre objets, il n'y a pas moyen de terminer proprement un contexte. Le départ d'un utilisateur est donc simulé par l'envoi d'un message d'adieu.

3.4 Le contexte serveur

Les tâches réalisées dans le contexte du serveur sont :

- fournir un mandataire aux clients et créer un mandant associé dans le contexte du serveur.
- effectuer la communication entre les mandants.
- gérer la table des utilisateurs connectés.

3.4.1 Exportation d'un mandataire

A l'initialisation, le serveur **speakServer** effectue les opérations suivantes :

- allouer un OID de groupe qui servira à l'ensemble des mandataires et mandants qui le contacteront.
- se déclarer au service de Nommage
- importer le code des mandataires

Par la suite, au moment d'une demande d'importation de mandataire par un client, la méthode **giveProxy** permettant d'exporter des mandataires, sera appelée par le service d'accointances (cf. 3.3.5). Cette méthode se chargera de :

- vérifier les droits d'accès de l'importateur
- la création du mandataire et de son mandant associé
- relier le mandant et le mandataire en mettant à jour leurs **trapReferences** respectives.
- associer l'OID du serveur comme OID de groupe
- déclarer le code à exporter dans le contexte client au service d'accointances

Le mandataire est alors migré dans le contexte client par le service d'accointances (cf. figure 3.3).

3.4.2 La gestion de la table des utilisateurs

La table des utilisateurs, gérée sous forme de liste chaînée, illustre la possibilité de données locales au serveur et aux mandataires. Cette table est distribuée à chaque mandataire, lorsqu'il se joint à l'application. Chaque mandataire s'occupe alors de la mise à jour de sa propre copie.

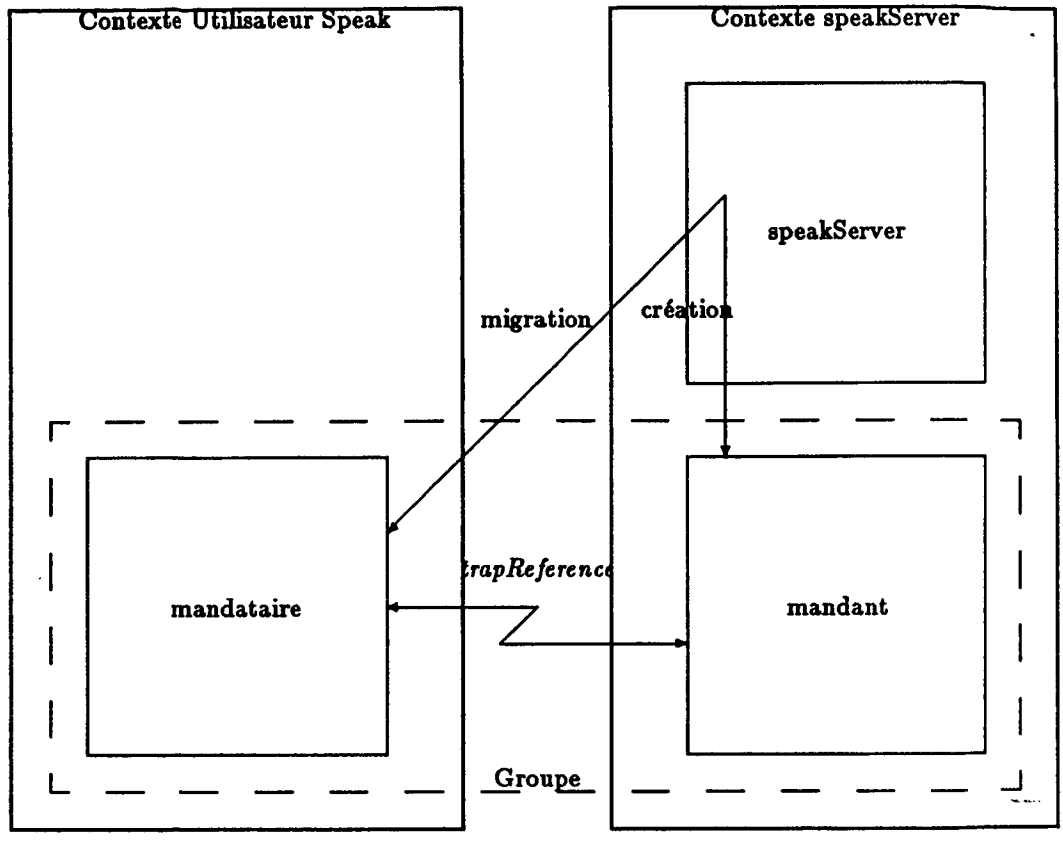


Figure 3.3 : Migration du mandataire speak

3.4.3 Communication entre utilisateurs

Fabriquation et reconfiguration de l'anneau entre mandants

Pour que les mandants puissent communiquer, le serveur doit s'occuper de les mettre en relation, c'est à dire, de fournir à chacun l'adresse d'un autre mandant. Un anneau est construit reliant chaque mandant à un seul voisin.

D'abord le serveur fournit l'adresse du premier mandant créé en paramètre au constructeur de chaque mandant.

Ensuite, ce nouveau mandant contacte le premier, en lui founissant sa propre adresse.

Le premier transmet cette adresse à son voisin, et ainsi de suite jusqu'au dernier, qui peut alors refermer l'anneau sur le nouveau mandant.

Ce tour complet permet la mise à jour du nouvel utilisateur dans la table de chaque mandant, mais impose au dernier mandant et à lui seul, de jouer un rôle particulier.

Lorsqu'un utilisateur désire quitter l'application, le protocole de fermeture comprend la reconfiguration de l'anneau. Le mandant voulant partir, transmet l'adresse du suivant à son précédent. La reconfiguration est donc indépendante du serveur, sauf dans le cas, où le dernier mandant désire quitter l'application.

Circulation d'un message dans l'anneau

Lorsqu'un mandataire effectue une cross-invocation, le mandant reçoit ce message sur son stub, et le transmet au mandant suivant. Chaque mandant compare le champs destinataire du message avec sa propre adresse (pour pouvoir cross-invoquer son mandataire en cas d'égalité), et transmet ce message dans tous les cas, jusqu'à ce que l'émetteur le retire. Ce tour complet n'est pas significatif dans la version 1, mais offre la possibilité d'implémenter des contrôles rigoureux (par exemple, des acquitements, ou la reconnaissance de la disparition d'un utilisateur).

3.5 Analyses et remarques sur speak sous SOS version 1

Taille des executables	en octets	en lignes de code C++
le noyau SOS	24500	1750
le service d'accointances	74000	2400
le service speakServer	82000	
speakServer (sans bibliothe'que)	10000	600
le programme speak	74000	
speak (sans bibliothe'que)	3000	350
le code du mandataire	3000	350

Ces chiffres indiquent le coût imposé par la taille des bibliothèques, et montrent notre intérêt pour les bibliothèques partagées et la mémoire partagée. Le nombre de lignes de code à écrire, important par rapport à la simplicité de l'application, s'explique par le manque d'outils de la première version de SOS (par exemple, les "cross-invocations" manuelles). Cette première application impose à l'implémenteur de définir tous ses protocoles. Dans l'avenir, des protocoles de base pourront être prédéfinis sous forme de bibliothèques, facilitant ainsi l'écriture possible de protocoles plus complexes.

Nous ne pouvons fournir de chiffres au sujet des performances : ce programme interactif ne peut donner une grandeur des coûts de l'importation d'un mandataire et des temps de communication. Néanmoins, l'efficacité fait entièrement partie des objectifs de SOS (langage compilé C++, effets de localité chez les mandataires).

Enfin, la rapidité de transformation de l'application descrite en 3.7, pour s'intégrer à la version 2 de SOS, donne une idée positive du temps de développement.

3.6 Conclusion sur speak version 1

Cet exemple a donc permis de démontrer certaines insuffisances de SOS Version 1 et de confirmer les principaux outils manquant à cette version :

- Tous les utilisateurs doivent se retrouver sur le site où est lancé le serveur **speakServer**. En effet, le service de communication distante n'est pas intégré à SOS Version 1, et par conséquent pas de service de Nommage réparti.
- Le serveur doit englober les fonctionnalités de distribution (multi-casting) entre les différents mandataires. En effet, la Version 1 ne comporte pas de protocole de diffusion et ne permet qu'une seule `trapReference` immuable par objet.
- L'implémenteur d'un service doit créer l'objet "code" du mandataire à l'exécution, car il n'existe pas de service de Stockage.

- Un utilisateur ne peut quitter proprement la conférence, car la destruction d'un contexte entraîne la terminaison du noyau SOS et par conséquent des autres contextes (dépendances entre objets non gérée).

Cette application n'est finalement pas répartie, à cause des limitations de la Version 1. L'application est néanmoins répartie dans sa conception, ce qui lui permet d'évoluer facilement.

3.7 L'intégration de speak dans SOS version 2

La prochaine version de SOS est prévue pour l'automne 87, et intégrera les services de Communication, de Nommage réparti, et de Stockage. Contrairement à la version 1 (figure 3.1), l'intégration du service de Communication distante à SOS version 2 permet de connecter directement les mandataires entre eux (figure 3.4). Les mandants crées par le serveur **speakServer** ne jouent donc plus aucun rôle dans cette communication. Ainsi, il suffit de remonter le code existant précédemment chez les mandants de la version 1, dans les mandataires de la version 2, puisque la gestion de l'anneau est identique. Ce paragraphe décrit les modifications de **speak**, pour prendre en compte les améliorations de SOS Version 2.

Ces modifications sont regroupées dans le code du mandataire, puisque d'une part ces modifications concernent la communication inter-contexte, et d'autre part, les changements dans l'implémentation d'une méthode ne modifient pas les appels sur cette méthode. Dans **speak** Version 2, le serveur **speakServer** crée la famille "speak-Family" et délivre aux mandataires, la possibilité de se joindre à cette famille, en ajoutant trois OIDs à la liste du mandataire :

- l'OID de groupe du service
- l'OID attribué au premier mandataire créé
- l'OID de membre de ce mandataire dans la famille

Le mandataire nouvellement crée peut alors obtenir ces OIDs en parcourant sa propre liste. Pour se connecter à la famille, les mandataires doivent importer un objet **speakRpc** dérivant de la classe "rpcMgrProxy", ayant pour effet l'importation d'un mandataire du service de Communication. Le mandataire de communication permet de prendre en compte la répartition de manière transparente à l'objet **speakRpc**, qui effectue de simples cross-invocations. La désignation du destinataire est réalisée en fournissant à la primitive **crossInvoke**, l'OID du membre à contacter. L'insertion d'un nouvel objet speakRpc est réalisée en communiquant son propre OID aux autres membres. Cet OID sert aussi à reconnaître le retour de son propre message.

Les relations entre les objets **speakRpc** sont identiques à celles des mandants de la version 1. La connaissance de l'OID attribué au premier mandataire permet la reconfiguration de l'anneau entre **speakRpc**, selon la méthode descrite dans 3.4.3.

L'utilisation des nouveaux services de la version 2 engrendre peu de modifications à l'application elle-même. Cette version ne possède pas les principaux défauts de la première version et atteint les objectifs fixés, lors de la conception de cette application. Toutefois, un utilisateur ne peut toujours pas quitter l'application indépendamment des autres.

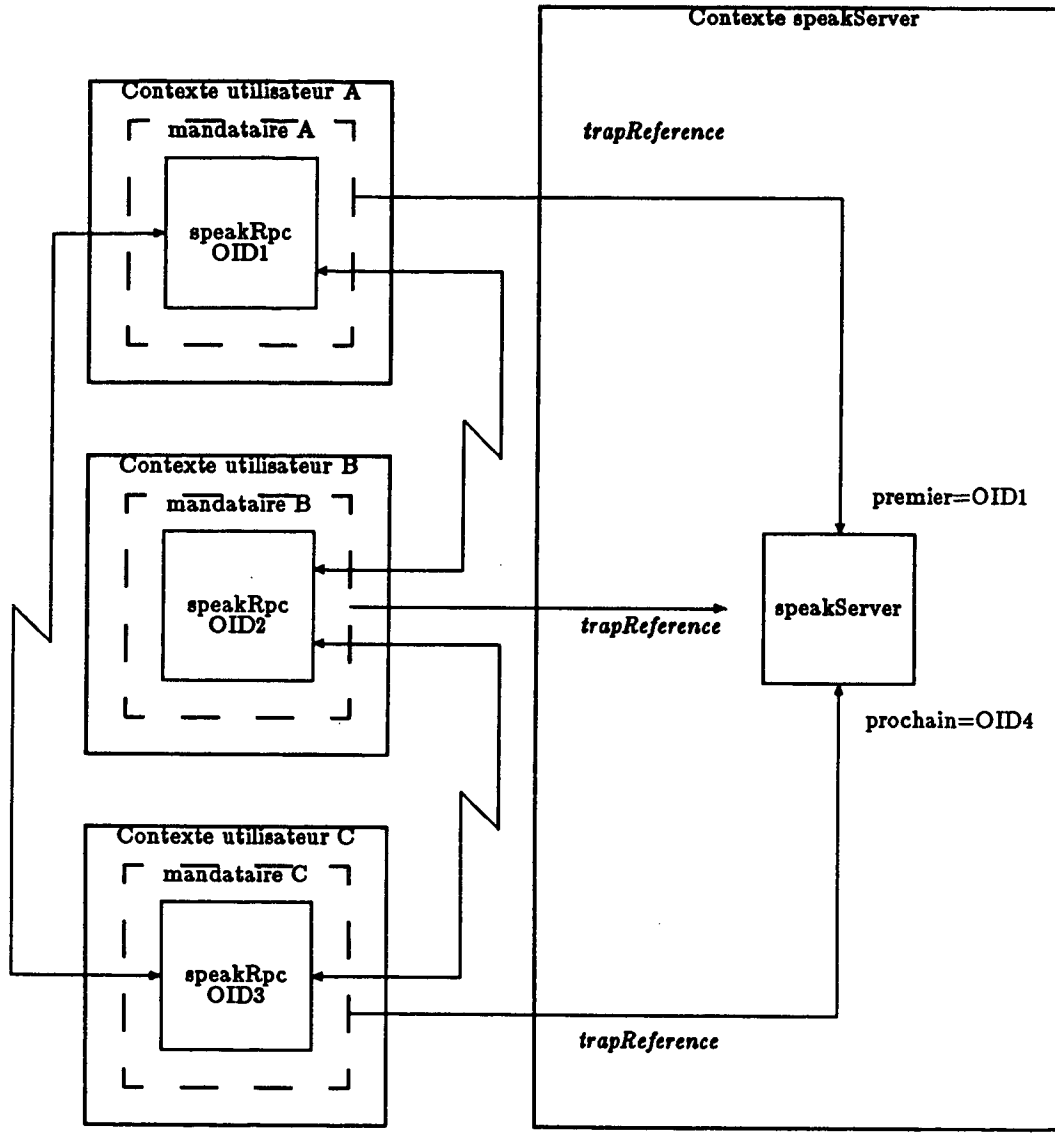


Figure 3.4 : modèle *réparti* réalisé pour SOS-V2

Chapitre 4

Edition de liens dynamique au format COFF

Le concept de mandataire impose la migration des données et du code de l'objet exporté, du contexte serveur vers le contexte client. Le service d'accointances et le noyau, qui effectuent cette tâche, ont donc besoin de pouvoir charger dynamiquement cet objet dans l'espace d'adressage du client. L'existence dans un tel système d'un éditeur de liens dynamique, est donc nécessaire.

La possibilité de choisir entre liaison statique et dynamique reste indispensable à cause du coût de la liaison dynamique réitérée à chaque exécution. L'image des exécutables présentent un gain de place sur disque et en mémoire. Chaque programme charge uniquement les modules qui dépendent de son propre environnement. La mise à jour des bibliothèques est de ce fait automatique, et le partage du code des bibliothèques [Arn86] paraît plus facile à mettre en œuvre.

Par la suite, nous distinguerons deux types de code :

- Le code *importateur*
- Le code *importé*

Le code *importateur* représente un programme exécutable, dont certaines routines ne sont pas encore chargées. Le code *importé* constitue la partie de code non définie dans le code importateur lors de la phase d'édition statique, et qui doit être importée dynamiquement lors de l'exécution.

L'édition dynamique comporte deux phases :

- La première, lors de l'exécution du programme importateur, la *détection* de la demande d'importation.
- La seconde, le *chargement* : trouver, puis charger le code *importé*, et enfin de le lier au code *importateur*.

La première phase est préparée lors de la compilation et n'apparaîtra pas toujours explicitement dans les exemples cités. C'est en effet la manière d'effectuer le chargement du code importé et de sa liaison au code importateur qui nous intéressent principalement.

Nous parlerons d'abord des différentes implémentations possibles, puis de la version réalisée pour SOS (Sun/4.2BSD) intégrée au compilateur C++, et enfin l'adaptation de cet éditeur de lien au format COFF, dans le cadre du portage de SOS sur Métaviseur [BULL].

4.1 Quelques mises en œuvre existantes

Ce chapitre présente deux exemples d'éditeur de liens dynamique, au-dessus de l'éditeur statique standard d'Unix BSD, proches de la version SOS.

Nous ne parlerons pas de l'exemple de Multics [Org72], car cette implémentation dépend du noyau (en particulier de la mêmoire virtuelle), et ne propose que des liaisons dynamiques.

4.1.1 Implémentation liée à l'éditeur de liens d'Unix 4.2BSD

Une réalisation courante du *chargement* dynamique est d'utiliser la possibilité de "chargement incrémental" de l'éditeur de liens `ld` d'Unix BSD.

Dans un premier temps, on prépare un fichier objet **importateur** incomplet et une version intermédiaire **intermédiaire** du code *importé* que nous appelons ici **importé** :

ld -A **importateur** -x -T **zone-allouée** -o **intermédiaire importé**

Cette dernière commande réalise l'édition de liens partielle. Le fichier **importé** est maintenant prêt à être chargé à l'adresse **zone-allouée** du programme **importateur**. C'est à ce dernier, pendant son exécution, de réaliser le chargement en lisant le fichier objet **intermédiaire** à son adresse **zone-allouée**. L'option "-x" permet de ne traiter que les symboles externes.

L'intérêt de cette démarche, est de ne pas se préoccuper de la relocalisation qui est faite par l'éditeur de liens statique. La table des symboles du code **importateur** est prise comme référence pour ajouter les symboles du code **importé**.

Cette option n'existe pas dans Système V, et notamment sur Métaviseur. Cette implémentation ne permet pas non plus, de référencer une procédure du code *importateur* par une procédure du code *importé*. Ces insuffisances limitent son utilisation malgré sa facilité de mise en œuvre.

4.1.2 Camphor

Camphor [Kaz86] a été réalisé au CMU dans le cadre du projet Andrew [Mor86], et constitue un environnement d'édition de liens dynamique pour programmes C.

Chaque code (*importateur* et *importé*) comprend une partie **déclaration** et une partie **définition** de l'interface à exporter (ou à importer). L'environnement Camphor comprend :

- un ensemble de macros du préprocesseur C, qui génèrent des appels indirects
- une bibliothèque de fonctions liées statiquement avec le code *importateur*, et effectuant l'édition dynamique

La *détection* se fait en récupérant le signal "Illegal Instruction", généré par Unix lorsque le programme essaie d'exécuter le code non défini (code *importé*), par une routine en assembleur. Cette routine fait partie de la bibliothèque Camphor. Celle-ci se charge :

- de trouver le nom du fichier à importer
- de charger le contenu de ce fichier dans la zone données de l'*importateur*
- de la relocalisation et du traitement des symboles indéfinis
- et enfin, de redémarrer l'instruction qui a provoqué le signal d'erreur, sachant que cette fois-ci le code est présent.

A cet effet, avant le déroutement du signal, il est nécessaire de récupérer le compteur ordinal dans la pile. Cette opération n'est pas réalisable de façon portable dans Système V.

Ce mécanisme, bien qu'il dépende peu de l'éditeur de liens statique d'Unix, reste dépendant machine, à cause des routines en assembleur s'occupant de la récupération du signal Unix. L'usage des macros limite aussi cette implémentation.

4.1.3 L'édition de liens dynamique dans SOS, intégrée à C++

La mise en œuvre pour SOS respecte les contraintes suivantes :

- respecter l'éditeur de liens standard d'Unix
- de ne faire aucune modification au noyau Unix
- être portable

Cette version, réalisée par P. Gautron, s'inspire de Camphor pour le *chargement*. La *détection* est intégrée au compilateur C++, grâce à une syntaxe particulière. La déclaration suivante :

```
dynamic class Foo foo;
```

indique que l'objet `foo` est une instance de la classe *dynamique* Foo, c'est à dire que son code sera lié dynamiquement. Le compilateur génère un appel à l'éditeur de liens, puis un appel indirect au constructeur de cette classe. L'éditeur retourne l'adresse du point d'entrée du code *importé*. Les appels aux routines *importées* se font par indirection, et les prochains appels ne passeront plus par l'éditeur de liens.

4.2 Portage de l'éditeur au format COFF

Après une présentation du format COFF d'Unix Système V, nous décrirons notre façon de charger et de lier du code au format COFF.

4.2.1 Format COFF

COFF divise les zones texte et données des exécutables en plusieurs "sections". Chaque exécutable comporte au moins trois sections nommées respectivement ".text", ".data" et ".bss". Cette structuration permet de réserver une section pour des usages privés. Par exemple, l'ajout d'un nouveau type de relocalisation permet de supporter l'adressage plus complexe de certains processeurs. A cet effet, une section ".tv", peut contenir non seulement des références directes ou relatives, mais aussi des *références indirectes*.

La table des symboles COFF contient des informations n'existant pas dans "a.out.h" d'Unix V7 ou 4.2BSD :

- le numéro de section où est localisé le symbole
- le type ou la définition du type dérivé (description des structures C)
- le nombre d'entrées auxiliaires (informations supplémentaires pour les symboles complexes)

Un binaire au format COFF contient le type des symboles, ainsi que le type des arguments passés à une fonction. Lors de l'importation dynamique de code, une vérification d'interface possible consisterait à comparer le type attendu par le code *importateur* avec le type des symboles *importés*.

De plus, l'entête des sections comportent des informations concernant le numéro de ligne dans le code source. La connaissance du type des symboles et le numéro de ligne référencé facilitent le "déboguage symbolique".

Enfin, une bibliothèque de fonctions permet d'accéder aux champs des structures de l'exécutable, de manière portable (contrairement aux macros définies dans "a.out.h")

Cependant, la complexité du format COFF ne permet pas une traduction facile des programmes au format "a.out.h". Peu de documents sont disponibles, et une phase d'apprentissage est nécessaire.

4.2.2 Description de l'éditeur de liens dynamique

Chargement

- Le chargement se passe de la manière suivante :
  - lire l'entête de l'exécutable
  - allouer la place pour lire la table des symboles, les zones texte et données et la table de relocalisation
  - parcourir la table des symboles et extraire le nom des symboles à partir de la table des chaînes.
  - noter au passage la taille à allouer pour la table des externes et celle des indéfinis.
  - puis effectuer une seconde passe, pour remplir ces tables.
- Les choix adoptés lors du portage sous COFF sont les suivants :

  Ajouter les informations relatives aux sections.

  Pour pouvoir, continuer à maintenir une cohérence avec l'éditeur au format "a.out.h", et reporter facilement les futurs changements, nous choisissons d'ajouter quelques redondances. Ainsi, les tailles des zones de relocalisation sont dupliquées (dans une table mémorisant les informations sur les sections et dans une variable que le code source utilise souvent).

  Dans un souci de portabilité,

  l'utilisation des fonctions de la bibliothèque "ld" a été choisie au dépend d'une perte probable de performances (exemple : la lecture des informations de relocalisation symbole par symbole).

Traitement des indéfinis et la relocalisation d'informations

Les indéfinis

Lors du parcourt de la table des symboles de l'exécutable *importé*, la valeur de chaque symbole indéfini est recherchée dans les exécutables chargés antérieurement, puis dans la table des bibliothèques. Si ce symbole est encore indéfini, la routine de traitement des erreurs est appelée.

La relocalisation

La relocalisation s'effectue en deux passes pour différencier les symboles du type texte de ceux du type donnée, mais la méthode est identique.

Les ajustements

Plusieurs ajustements sont possibles selon le type du symbole. Dans le cas des variables statiques, pas de relocalisation nécessaire. Dans le cas des variables externes, il suffit d'ajouter au déplacement précédemment calculé, la valeur donnée par la table de relocalisation du symbole. Enfin, dans le cas d'une variable "relative au compteur ordinal", l'adresse de la zone allouée doit être ajoutée à ce déplacement.

Comme le montre la réalisation décrite en 4.1.1, utilisant la possibilité de chargement incrémental de l'éditeur de liens statique, la relocalisation dans la version 4.2BSD est similaire, qu'elle soit statique ou dynamique. Ce n'est pas le cas dans Système V. La relocalisation de l'éditeur dynamique est donc un mixage de la version 4.2BSD et de la version de l'éditeur statique de Système V. Ainsi, dans COFF, certains types de symboles dépendent de la relocalisation de symboles auxiliaires, qui sont statiques, et donc non conservés dans la version 4.2BSD. Par exemple, pour reloger une variable de type chaîne de caractère, il faut aussi reloger une variable statique (".data" par exemple), afin de reloger le contenu de l'adresse référencée.

La seconde annexe montre un exemple d'importation.

4.3 Conclusion

Le portage de cet éditeur de liens sous système V, ne comprend pas pour l'instant les modifications au compilateur C++, et n'est donc pas opérationnel. Cependant, de nombreux tests (à la manière du premier exemple d'éditeur de liens dynamique sous Unix) ont permis de charger et d'éditer dynamiquement de nombreux cas possibles, permettant ainsi d'envisager le reste du portage de SOS sur Métaviseur. De plus, le format COFF est un atout pour apporter une solution à certains problèmes existants tels que les bibliothèques partagées ou la vérification de types lors de l'importation dynamique.

Ce portage satisfait les objectifs suivants :

- La portabilité

  L'usage des fonctions de la bibliothèque COFF permet une compilation plus facile sur toutes les machines tournant Système V et de ne pas se limiter au Métaviseur.

- La compatibilité entre versions

Des informations redondantes sont gardées afin que le source des deux versions reste compatible et de permettre une intégration rapide des futures versions. Le recul, consistant à traduire les "sections" du format COFF en un équivalent dans le format "a.out.h", sera corrigé dans la version définitive.

- L'efficacité

  Le travail effectué conserve toutes les astuces pour accélérer le mécanisme de liaison dynamique : réduction des appels systèmes, regrouper les allocations mémoires et lectures sur disque, tenir compte des possibilités d'Unix (favoriser les lectures séquentielles et utilisation de la localisation d'informations dans le cache). D'autres optimisations spécifiques à COFF sont prévues.

Chapitre 5

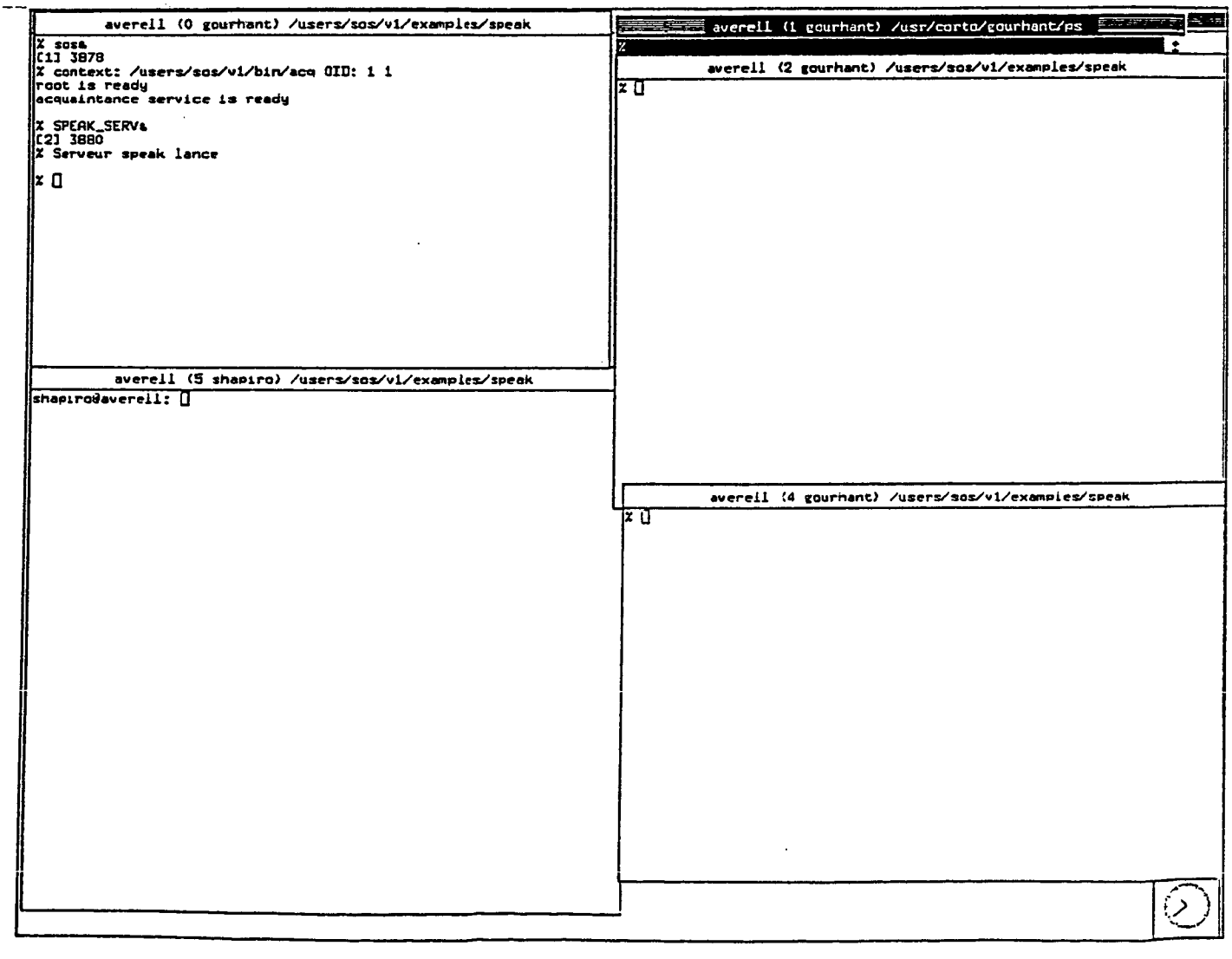
Conclusion

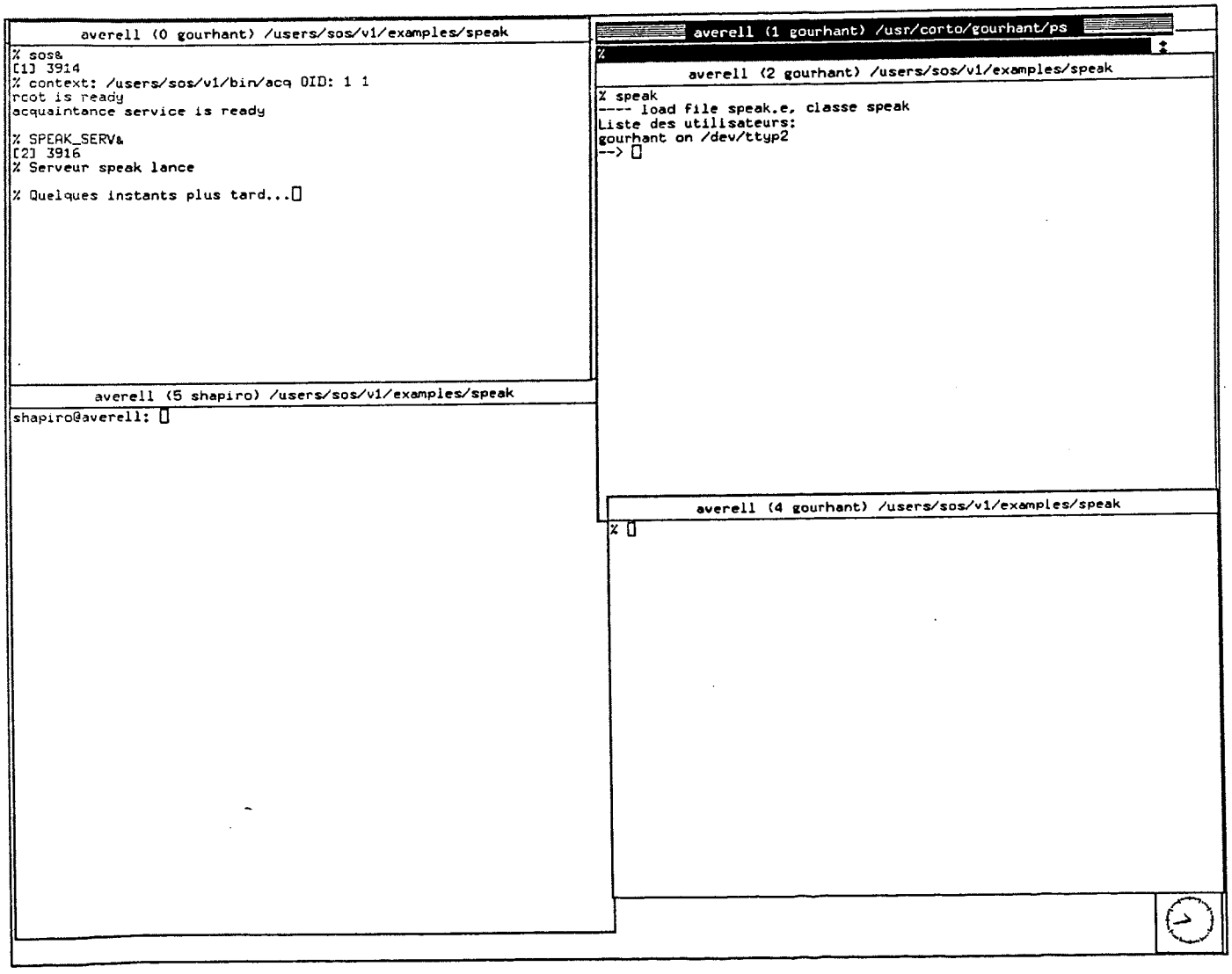
La première version de l'application **speak** n'a pas atteint les objectifs fixés, mais a permis de mettre en valeur certaines insuffisances de SOS version 1 (cf 3.6). Le principal obstacle est l'absence de multi-casting à l'intérieur d'un groupe. Les objectifs initiaux sont atteints par la version 2, qui est entièrement répartie. Les changements effectués pour passer à la version 2 illustrent l'encapsulation d'interface dans le mandataire et la réutilisabilité des modules (transfert du code des mandants dans les mandataires). Cependant, la solution de certains problèmes est repoussée aux futures versions : en particulier, l'absence de gestion des dépendances entre objets ne permet pas de quitter proprement l'application.

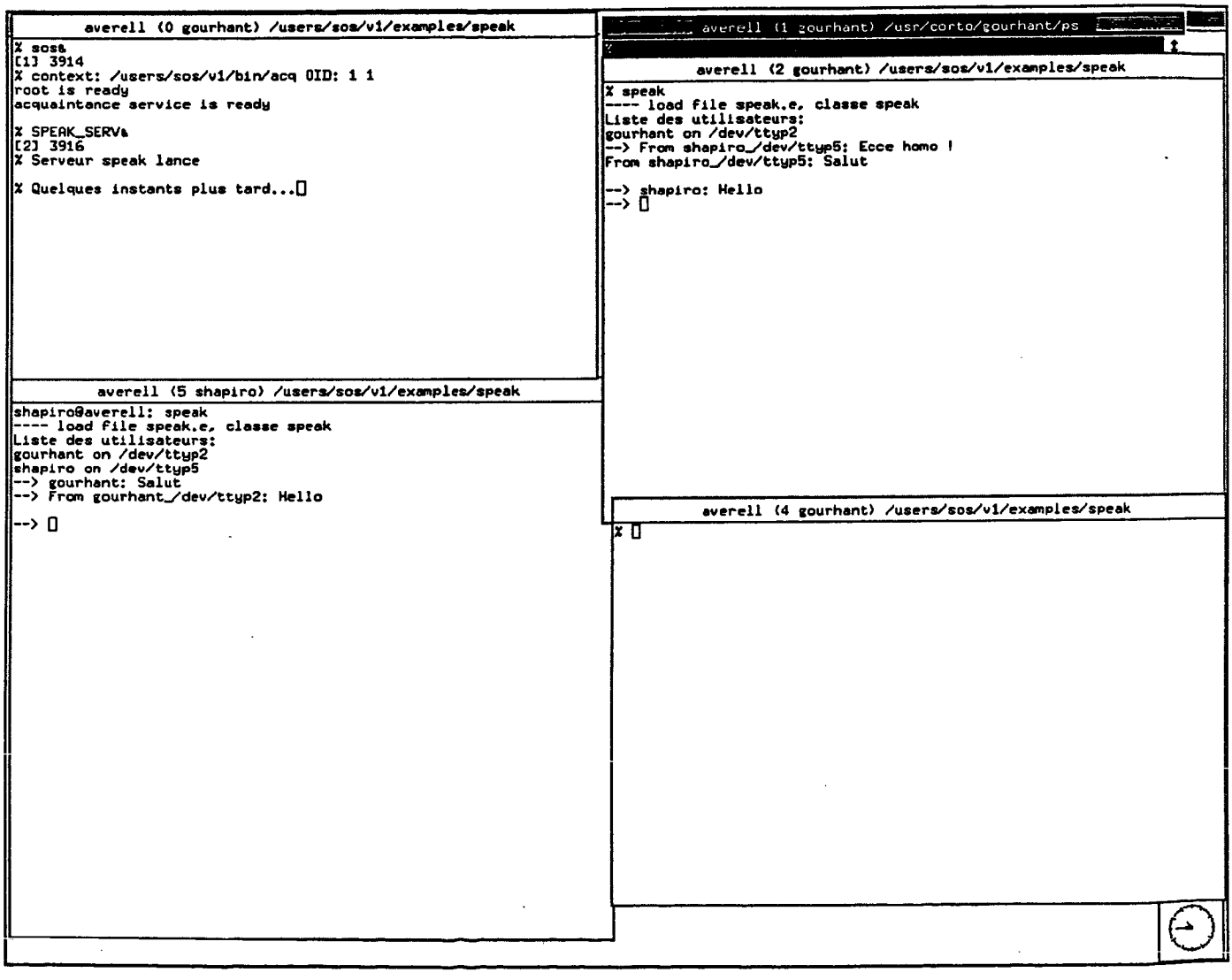
Le portage de l'éditeur dynamique de liens au format COFF permet d'envisager celui de SOS sur le Métaviseur et des solutions aux outils dont ont besoin les futures versions de SOS. Par exemple, les informations sur le type des symboles du code importé facilite une vérification d'interface lors de l'édition dynamique de liens. Enfin, c'est aussi au moment de l'exécution que l'on peut traiter du problème des bibliothèques partagées.

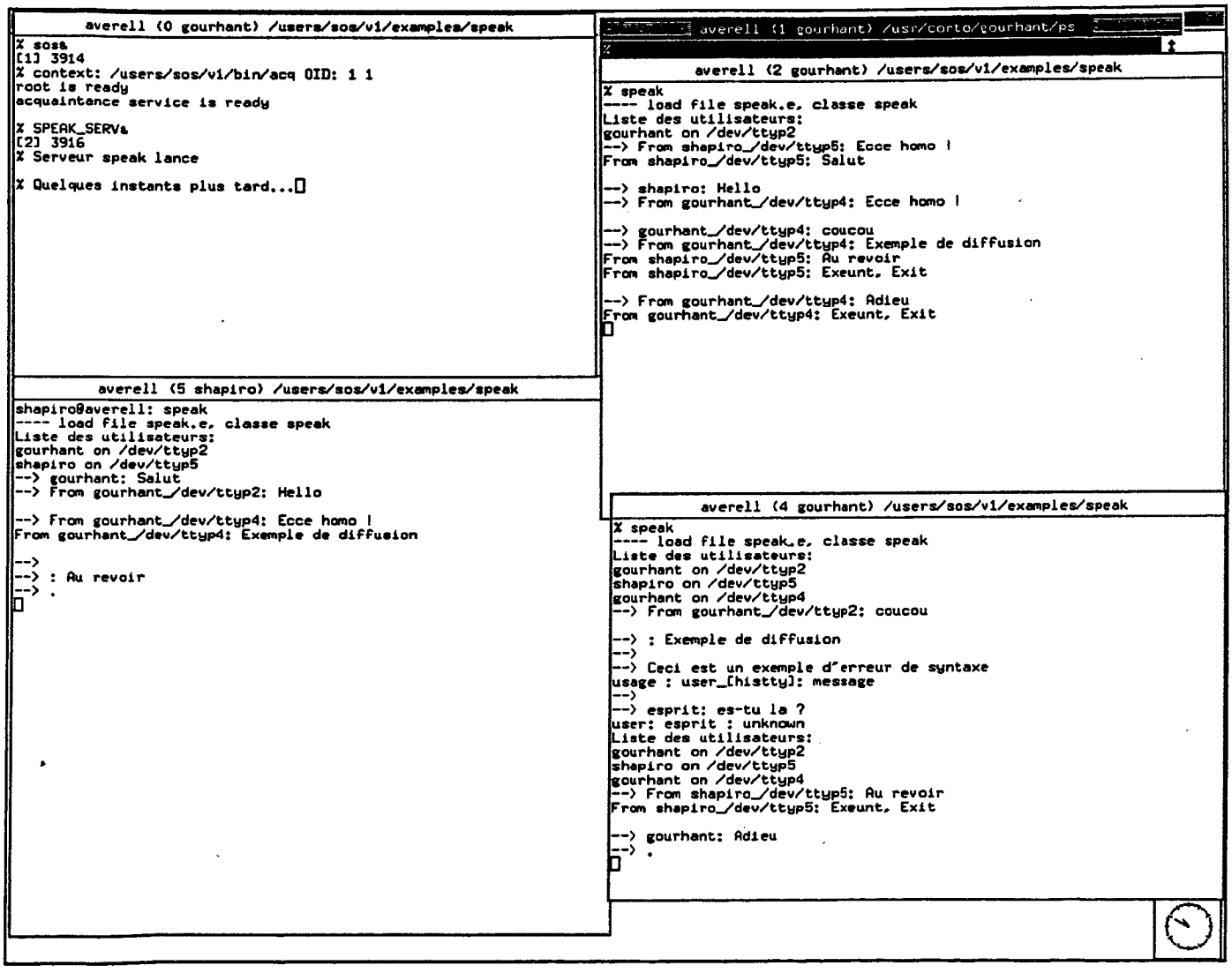
Annexe A

Exemple d'utilisation de speak




averell (0 gourhant) /users/sos/v1/examples/speak
% sos&
[1] 3914
% context: /users/sos/v1/bin/acq OID: 1 1
root is ready
acquaintance service is ready
% SPEAK_SERV&
[2] 3916
% Serveur speak lance
% Quelques instants plus tard...
averell (1 gourhant) /usr/corto/gourhant/ps
%
averell (2 gourhant) /users/sos/v1/examples/speak
% speak
---- load file speak.e, classe speak
Liste des utilisateurs:
gourhant on /dev/ttyp2
-->
averell (5 shapiro) /users/sos/v1/examples/speak
shapiro@averell:
averell (4 gourhant) /users/sos/v1/examples/speak
%


averell (0 gourhant) /users/sos/v1/examples/speak
% sos&
[1] 3914
% context: /users/sos/v1/bin/acq OID: 1 1
root is ready
acquaintance service is ready
% SPEAK_SERV&
[2] 3916
% Serveur speak lance
% Quelques instants plus tard...
averell (1 gourhant) /usr/corto/gourhant/ps
averell (2 gourhant) /users/sos/v1/examples/speak
% speak
---- load file speak.e, classe speak
Liste des utilisateurs:
gourhant on /dev/ttyp2
--> From shapiro_/dev/ttyp5: Ecce homo !
From shapiro_/dev/ttyp5: Salut
--> shapiro: Hello
--> []
averell (5 shapiro) /users/sos/v1/examples/speak
shapiro@averell: speak
---- load file speak.e, classe speak
Liste des utilisateurs:
gourhant on /dev/ttyp2
shapiro on /dev/ttyp5
--> gourhant: Salut
--> From gourhant_/dev/ttyp2: Hello
--> []
averell (4 gourhant) /users/sos/v1/examples/speak
% []


averell (0 gourhant) /users/sos/v1/examples/speak
% sos&
[1] 3914
% context: /users/sos/v1/bin/acq OID: 1 1
root is ready
acquaintance service is ready
% SPEAK_SERV&
[2] 3916
% Serveur speak lance
% Quelques instants plus tard...
averell (1 gourhant) /usr/corto/gourhant/ps
averell (2 gourhant) /users/sos/v1/examples/speak
% speak
---- load file speak.e, classe speak
Liste des utilisateurs:
gourhant on /dev/ttyp2
--> From shapiro_/dev/ttyp5: Ecce homo !
From shapiro_/dev/ttyp5: Salut
--> shapiro: Hello
--> From gourhant_/dev/ttyp4: Ecce homo !
--> gourhant_/dev/ttyp4: coucou
--> From gourhant_/dev/ttyp4: Exemple de diffusion
From shapiro_/dev/ttyp5: Au revoir
From shapiro_/dev/ttyp5: Exeunt, Exit
--> From gourhant_/dev/ttyp4: Adieu
From gourhant_/dev/ttyp4: Exeunt, Exit
averell (5 shapiro) /users/sos/v1/examples/speak
shapiro@averell: speak
---- load file speak.e, classe speak
Liste des utilisateurs:
gourhant on /dev/ttyp2
shapiro on /dev/ttyp5
--> gourhant: Salut
--> From gourhant_/dev/ttyp2: Hello
--> From gourhant_/dev/ttyp4: Ecce homo !
From gourhant_/dev/ttyp4: Exemple de diffusion
-->
--> : Au revoir
--> .
averell (4 gourhant) /users/sos/v1/examples/speak
% speak
---- load file speak.e, classe speak
Liste des utilisateurs:
gourhant on /dev/ttyp2
shapiro on /dev/ttyp5
gourhant on /dev/ttyp4
--> From gourhant_/dev/ttyp2: coucou
--> : Exemple de diffusion
-->
--> Ceci est un exemple d'erreur de syntaxe
usage : user_[histtyl: message
-->
--> esprit: es-tu la ?
user: esprit : unknown
Liste des utilisateurs:
gourhant on /dev/ttyp2
shapiro on /dev/ttyp5
gourhant on /dev/ttyp4
--> From shapiro_/dev/ttyp5: Au revoir
From shapiro_/dev/ttyp5: Exeunt, Exit
--> gourhant: Adieu
--> .

NAME

speak – on-line communication with other users

SYNOPSIS

speak

DESCRIPTION

speak implements a distributed on-line conference.

Any user can join a conference to exchange messages. **speak** is invoked without arguments. It is ready to accept commands when it gives you the list of users and prints its prompt ("speak>").

At this stage, if you wish to send a message to someone else, use the following format:

speak> *user* : the message

user is of the form:
logname [@hostname][_ttyname]

logname is the login name of a user who has joined the conference, and specifies the receiver of the message.
If you want to speak to a user who is recorded more than once, the *hostname* or the *ttyname* arguments may be used to discriminate the receiver.
A empty *logname* means that the message is broadcasted to all connected users.

BUGS

(1) The server **speakServer** must be active.

(2) To exit, just type ^C. This terminates the whole SOS and all running contexts.

(3) Only 8 users per host.

SEE ALSO

intro(1), sos(1), speakServer(1)

NAME

speakServer – the server of speak

SYNOPSIS

speakServer

DESCRIPTION

speakServer is a server used by the **speak** program.

BUGS

It is possible to have only one conference active at one time

SEE ALSO

speak(1)

Annexe C

Manuel de référence l'éditeur de liens dynamique (SOS Version 1)

NAME

dynamic – introduction to dynamic classes in SOS

SYNOPSIS

```
ref* r;
importRequest* ir;
char *srchName1, *srchName2;

dynamic class [(r )] clName {
   // class declaration
};
clName* i1 = new clName;
clName* i2 = new dynamic [(r )] clName (ir);

dynamic class [(srchName1,srchName2... )] clName {
   // class declaration
};
clName* i1 = new clName;
clName* i2 =
        new dynamic [(srchName1,srchName2... )] clName (ir);
```

DESCRIPTION

The keyword *dynamic* in a class declaration means a dynamic class. The code of this class will be loaded at the first instanciation. Instance declarations can be static (*i1*) or dynamic (*i2*). Data allocation is local in the former case, imported with code in the latter case.
Type of dynamic declarations may be *ref** (see *reference*(2)) or *char**. In this second case, an extra call to *sosLookup* (2) is generated.
In SOS, dynamic class must own at least one constructor with the first argument of type *importRequest** (or derived) - see: "On the use of the dynamic link editor in SOS v2" -.

SrchName or *r* are optional; they are interpreted differently under Unix and under SOS:

- In Unix, *srchName* is directly the name of a file containing the code for the class (*r* has no sense)
- In SOS, *r* is the reference of a principal to be queried for a proxy. The principal must have previously registered this name with the Name Service. The principal must create an object of class code, with an indication of the file containing the code for the class (see *code(2)*). The proxy is imported and becomes the new instance.

There is no default name, so a null name can raise an exception. The code must be found in a file previously compiled by **makexport(1)**; it must be in a directory named in the Unix environment variable **DLPATH.**

The actual migration and/or loading of code is performed by the procedures *sosFindCode* or *sosImport* They are called at instantiation time, before the call to the local class constructor, and before:

- any instruction in the block for an **auto** variable;
- allocation for an object allocated by **new**;
- the execution of **main()** for static variables.

EXAMPLE

```
// in the header file
dynamic struct A {
  A ();
  A ( importRequest* );
  int m ();
};
```

```
// in the export file, compiled with makexport
A::A () {
  printf ("constructor reached\n");
}

A::A ( importRequest* ir ) {
  ...
}

A::m () {
  return printf ("method m reached\n");
}

// in the import file, compiled with CC -DL
main () {
  ref* r;
  importRequest* ir;
    ...
  A* a1 = new dynamic (r) A (ir);
  a1 -> m ();

  A a2;
  a . m();
}
```

FILES

file.c	source file
file.e	export file

SEE ALSO

sosCC(1), code(2), importRequest(2), makexport(1), reference(2).
Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley 1986.
B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall 1978.
On the use of the dynamic link editor in sos v2 in ~sos/v2/doc/dynalink

BUGS

The present version does not check the consistency of imported code with the importer; carefully check that the importer and the exporter were compiled using the same header files.

The code for a dynamic class is imported only once; hence it is useless to use different .e file names for subsequent instantiations.

If the dynamic linker can't find a procedure, the whole SOS halts with an error message when the procedure is called.

DIAGNOSTICS

Dynamic link errors (and internal bugs) begin with the message "Error dynamic link".

NAME

makexport – C++ compilation of a dynamic class definition

SYNOPSIS

makexport [**-e** *export_file*] [*options*] *file*

DESCRIPTION

Makexport compiles a C++ dynamic class *definition*. The result of the compilation is a file in export format, suitable for subsequent dynamic loading. This should be placed in a directory named in your Unix environment variable DLPATH. **Makexport** uses CC with dynamic linking.

All *options* are passed verbatim to CC, except:

-e interpreted by *makexport* as the name of the export file. The default is the name of the file argument with suffix .c replaced by .e.

-g not supported by makexport.

FILES

file.c	input file
file..c	temporary cfront output
file.o	object file
file.e	export file

SEE ALSO

sosCC(1), dynamic(1)
Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley 1986.
B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall 1978.
On the use of the dynamic link editor in sos v2 , in ~sos/v2/doc/dynalink

BUGS

The present version does not check the consistency of imported code with the importer; carefully check that the importer and the exporter were compiled using the same header files.

The code for a dynamic class is imported only once; hence it is useless to use different .e file names for subsequent instantiations.

DIAGNOSTICS

Dynamic link errors (and internal bugs) begin with the message "Error dynamic link".

Annexe D

Exemple d'édition de liens dynamique dans SOS Version 1

```
$ cat ../essai.c
#include <stdio.h>

typedef int (*pint)();
pint ptr;
pint dynamic_link(...);
void dynamic_link_init(int =0);

main( int argc, char ** argv )
{
    int res;

    dynamic_link_init();

    // chargement et edition de liens des 2 executables fournis en argument

    ptr = dynamic_link( 0,"FOO",0,0,0,0,argv[1], argv[2], 0 );

    // ptr = l'adresse de chargement du premier binaire
     + l'offset deplacement pour acceder au point d'entre

    fprintf(stderr,"essai: execution du code\n");
    res = (*ptr)();
    printf("foo retourne: %d\n", res );
}
```

```
$ cat essai1_load.c
#include <stdio.h>
extern char** environ;        // environnement du Shell courant

typedef int (*func)();

func foo()
{
    int bar();

    fprintf( stderr, "foo atteint %s\n", environ[0] );

    return bar;
}

int
bar(){

    fprintf( stderr,"bar atteint\n" );
    return 4;
}

$ cat essai2_load.c

#include <stdio.h>
extern char** environ;

bar();

int foo2 (){
    fprintf( stderr,"foo2 atteint %s\n", environ[0] );
    return bar2();
}

bar2(){
    fprintf( stderr,"bar2 atteint\n" );
    return foo();
}

$ CC -o essai essai.c -ldynamic -lld

$ makexport essai1_load.c

$ makexport essai2_load.c

$ ls
essai    essai1_load.c essai2_load.c
essai.c essai1_load.e essai2_load.e
```

```
$ DLPATH=.

$ essai essai2_load.e essai1_load.e
---- load file essai2_load.e, classe FOO
load_file return= (0x46afc 289532), entry= (0x0 0)
---- load file essai1_load.e, classe FOO
load_file return= (0x4db50 318288), entry= (0x0 0)
code ptr = 289532 0x46afc
essai: execution du code
foo2 atteint DISPLAY=unix:0
bar2 atteint
foo atteint DISPLAY=unix:0
foo retourne: 318354
$
```

Table des matières

Références

[Alm84] G. Almes et C. Holman. Edmas : an object-oriented, locally distributed mail system. décembre 1984. University of Washington.

[Alm85] Guy Almes, Andrew Black, Edward Lazowska, et Jerry Noe. The Eden system: a technical review. *IEEE Transactions on Software Engineering*, SE-11(1), janvier 1985.

[Arn86] James Q. Arnold. Shared libraries on Unix System V. Dans *Summer Usenix Conference*, Atlanta (USA), juin 1986.

[Ber84] E. J. Berglund et D. R. Cheriton. Amaze: a distributed multi-player game program using the distributed V Kernel. Dans *Proc. 4th. Int. Conf. on Dist. Computing Syst.*, San Francisco, CA (USA), mai 1984.

[Bir85] K. Birman, W. Dietrich, A. El Abbadi, et T. Joseph. An overview of the ISIS project. *Newsletter of the IEEE Special Interest Group on distributed Computing*, juin 1985.

[Bla85] A. P. Black. Supporting distributed applications: experience with Eden. Dans OSR ACM, éditeur, *10th ACM Symposium on Operating System Principles*, pp. 2–12, Orcas Island WA (USA), décembre 1985.

[Bla86] A. Black, N. Hutchinson, E. Jul, et H. Levy. Object structure in the emerald system. Dans *ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, octobre 1986.

[Cal87] L.A. Call, D.L. Cohrs, et B.P. Miller. Clam — an open system for graphical user interfaces. Dans *OOPSLA'87*, Orlando, Florida (USA), octobre 1987.

[Che84] David R. Cheriton. The V-Kernel, a software base for distributed systems. *IEEE Software*, 1(2):19–42, avril 1984.

[Jon79] A. K. Jones. The object model: a conceptual tool for structuring software. Dans R. Bayer, R. M. Graham, et G. Seegmuller, éditeurs, *Operating Systems, an Advanced Course*, Springer-Verlag, New York (USA), 1979.

[Kaz86] Michael Leon Kazar. Camphor: a programming environnement for extensible systems. 86. Information Technology Center, address Carnegie-Mellon University.

[Ker84] B.W. Kerninghan et Rob Pike. *The Unix Programming Environment.* Prentice-Hall, 1984.

[Lam79] B. Lampson et R. F. Sproull. An open operating system for a single-user machine. Dans *Proc. 7th Symp. on O.S. Principles*, pp. 98–105, 1979.

[Lea83] P.J. Leach et al. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, 1(5):842–857, novembre 1983.

[Lis82] Barbara Liskov. Abstraction mechanisms in CLU. Communications of the ACM, 1982.

[Lis83] Barbara Liskov et Robert W. Scheiffler. Guardians and actions: linguistic support for robust distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3), juin 1983.

[McK85] Martin S. McKendry. Clouds: a fault-tolerant distributed operating system. *IEEE Tech. Com. Distributed Processing Newsletter*, SI-2(6), juin 1985.

[Mey87] Bertrand Meyer. Reusability: the case for object-oriented design. *IEEE software*, 4(2):50–64, mars 1987.

[Mor86] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosental, et F. D. Smith. Andrew: a distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, mars 1986.

[Org72] E.I. Organick. *The Multics system: an examination of its structure.* MIT Press, Cambridge, Mass. (USA), 1972.

[Ras81] R. Rashid et G. Robertson. Accent: a communication-oriented network operating system kernel. Dans *Proc. 8th Symp. on Operating Syst. Principles*, pp. 64–75, Asilomar Conference Grounds, Pacific Grove CA (USA), décembre 1981.

[Sha86a] Marc Shapiro. Structure and encapsulation in distributed systems: the Proxy Principle. Dans *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pp. 198–204, IEEE, Cambridge, Mass. (USA), mai 1986.

[Sha86b] Marc Shapiro et Sabine Habert. Un système d'exploitation orienté objets pour SOMIW. Dans *3èmes Journées d'Étude Langages Orientés Objet*, AFCET, Paris (France), janvier 1986.

[Sha87a] Marc Shapiro, Vadim Abrossimov, Philippe Gautron, Sabine Habert, et Mesaac Mounchili Makpangou. *Un recueil de papiers sur le système d'exploitation réparti à objets SOS.* Rapport Technique no. 84, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), mai 1987.

[Sha87b] Marc Shapiro, Vadim Abrossimov, Philippe Gautron, Sabine Habert, et Mesaac Mounchili Makpangou. SOS : un système d'exploitation réparti basé sur les objets. *Techniques et Sciences Informatiques*, 6(2):166–169, 1987.

[Str82] Bjarne Stroustrup. *A set of C classes for co-routine style programming.* Rapport no. CSRT 90, ATT, Murray Hill NJ (USA), 1982.

[Str85] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, 1985.

[Wal83] B. Walker, G. Popeck, R. English, C. Kline, et G. Thiel. The Locus distributed operating system. Dans *9th ACM Symposium on Operating System Principles*, pp. 49–70, octobre 1983.

[Zim84] Hubert Zimmermann, Marc Guillemont, Gérard Morisset, et Jean-Serge Banino. *Chorus: a Communication and Processing Architecture for Distributed Systems.* Rapport de Recherche no. 328, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), septembre 1984.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique